Coping with Inconsistent Models of Requirements

Tiihonen, Juha

Rheinisch-Westfaelische Technische Hochschule Aachen

2019

Tiihonen , J , Raatikainen , M , Myllyaho , L , Lüders , C M & Männistö , T 2019 , Coping with Inconsistent Models of Requirements . in L Hotz , M Aldanondo & T Krebs (eds) , Proceedings of the 21st Configuration Workshop Hamburg, Germany, September 19th to 20th, 2019 . CEUR Workshop Proceedings , vol. 2467 , Rheinisch-Westfaelische Technische Hochschule Aachen , Aachen , pp. 1-8 , Configuration Workshop , Hamburg , Germany , 19/09/2019 . < http://ceur-ws.org/Vol-2467/paper-01.pdf >

http://hdl.handle.net/10138/307682

Coping with Inconsistent Models of Requirements

Juha Tiihonen[1] and **Mikko Raatikainen**[1] and **Lalli Myllyaho**[1]
and **Clara Marie Lüders**[2] and **Tomi Männistö**[1]

Abstract. Issue trackers are widely applied for requirements engineering and product management. They typically provide good support for the management of individual requirements. However, holistic support for managing the consistency of a set of requirements such as a release is largely missing. The quality of issue data may be insufficient for global analyses supporting decision making. We aim to develop tools that support product management and requirement engineering also in cases where the body of requirements, e.g. for a software release, is inconsistent. Software releases can be seen as configurations of compatible, connected requirements. Our approach described in this paper can identify inconsistent elements in bodies of requirements and perform diagnoses using techniques from Knowledge Based Configuration. The research methodology follows the principles of Design Science: we built a prototype implementation for the approach and tested it with relevant use cases. The Qt Company has large sets of real requirement data in their Jira issue tracker. We characterize that data and use it for empirical performance testing. The approach can support product management and requirements engineering in contexts where large, inconsistent bodies of requirements are typical. Empirical evaluation shows that the approach scales to usage in large projects, but future work for improving performance is still required. Value in real use is highly plausible but demonstration requires tighter integration with a developed visualization tool, which would enable testing with real users.

1 Introduction

Over the years Issue *trackers* have become important tools to manage data related to products. The trackers are especially popular in large-scale, globally distributed open source projects [4, 5], such as Bugzilla for Linux, Github tracker for Spring Boot, and Jira for Qt. A tracker can contain thousands of bugs and other issues reported by different stakeholders. These issues typically become *requirements* for a future release of a product. These requirements are often related to each other—it is not uncommon to have the same requirement more than once thus being related as similar or duplicate; or one requirement requires another requirement. However, trackers primarily provide support for individual requirements over their life cycle. Even though dependencies can sometimes be expressed for each individual requirement, more advanced understanding or analysis over all issues and their dependencies in a system is not well supported: Developers do not conveniently see related requirements; a requirement engineer cannot deal with requirements as an interconnected entity; and a product manager does not see what requirements and issues are related to the requirements planned for the next releases. To aggravate the problem, the data in a tracker is heterogeneous and often inconsistent. Thus, trackers are not optimal for the concerns of product management or requirements engineering that need to deal with different requirement options, alternatives, and constraints, as well as their dependency consequences when deciding what to do or not to do. This lack of support exists despite dependencies are found to be one of the key concerns that need to be taken into account in requirements prioritization [7, 1, 17] and release planning [16, 2].

Our objective is to help holistic management of requirements while issue trackers are utilized. The specific focus is on the application of technologies common in the field of Knowledge Based Configuration *(KBC)* to support the stakeholders who are required to deal with dependent requirements and issues in a tracker in their daily work. We support decision making such as configuration of release plans instead of automating it as needs are not known well enough and criteria are hard to formalize. We describe the technical approach of a system that aims to provide such support. The system is based on generating a *requirement model* that closely resembles a traditional configuration model. We also provide data which in practice shows that the approach fits in the context and scales even to large projects.

We aim to address the following research questions: What are the major requirements of the system? What are the characteristics of real requirements data? How does the performance of computation scale up? The applied research methodology follows Design Science in the sense that the aim is to innovate a novel approach and bring it into a specific new environment so that the results have value in the environment [10, 19]. The context of research has been the Horizon 2020 project OpenReq[3]. Our primary case has been the Qt Company (see Section 3) with a large database of issues.

Previous work: Dependencies in Requirements In the field of requirements engineering research, both industrial studies [11, 18], and release planning [16] and requirements prioritization [17] methods emphasize importance of dependencies but lack details for the semantics of dependencies. However, taxonomies have been proposed for requirements dependencies [13, 3, 6, 20]. These include structural dependencies, such as *refines* or *similar*; constraining dependencies, such as *require* or *conflict*; and value-based dependencies, such as *increases value* or *increases costs*. Although the taxonomies share similarities with each other, the taxonomies vary in terms of size and clarity of dependency semantics. Only a few taxonomies have been studied empirically so that saturated evidence for an established or general taxonomy has not emerged.

This paper is structured as follows. Section 2 introduces our approach for holistically supporting requirement management and describes the system we developed. Section 3 describes the real indus-

[1] University of Helsinki, Finland, email: {juha.tiihonen, mikko.raatikainen, tomi.mannisto, lalli.myllyaho}@helsinki.fi
[2] University of Hamburg, Germany, email:lueders@informatik.uni-hamburg.de

[3] https://openreq.eu/

trial context applied for evaluation. Performance testing in Section 4 covers both the approach and results. These results are analysed in Section 5. Discussion forms Section 6. Finally, Section 7 concludes.

2 Approach & System

Software releases or a set of requirements can be seen as configurations of mutually compatible requirements. Consistency check and diagnosis techniques that are commonplace in KBC can be applied in the context of software release planning.

2.1 Context: Characteristics of issue tracker data

The characteristics of requirements in a tracker have a profound effect on a practical approach. The requirements are manually reported by different people—the granularity, level of detail, and quality differ. These differences would remain even if all issues had been manually reviewed, as e.g. in the case of Qt's triage process that can even send an issue back for further information or clarification. Even the typology or purpose of issues, such as epics for feature requests and bugs for deficiencies, is not always adhered to. Duplicated or similar issues are not uncommon: A bug or feature request can be reported by several persons, each possibly providing some unique characteristics or details that need to be preserved. The semantics of dependencies between requirements is not always completely clear and the dependencies are not applied consistently by different people. The relationships are not even necessarily marked at all. As a result, the data in a tracker is in practice doomed to be inconsistent and incomplete. Therefore inference on the whole database is difficult or even meaningless. Correcting the whole database is practically hopeless or at least impractical. Therefore, we believe it is more fruitful to provide requirement engineering with tools that can help to cope with the less-than-perfect data.

2.2 Conceptualization of the problem

If the whole tracker database is likely to remain inconsistent, could we restrict the focus to some relevant subsets? Our approach is based on this idea. We support *analyzing a requirement and its neighbourhood*. A requirement is taken to the point of focus. We follow any relationships (described below) of that issue to neighbour issues. A *transitive closure* of issues within desired *depth* is calculated as a graph. Depth is the minimal distance between two issues. The transitive closure is used as the *context for analyses*. Another natural context of analysis is a *release*. An issue can be assigned to a specific release such as 4.12.1. The combined neighbourhoods of the issues of a release can be taken as the context of analysis[4]. Consequently, for a given context of analyses, a *requirement model is dynamically generated*. The requirement model is then mapped (through several layers) into a formal model that supports inference. We combine inference with procedural analysis of inconsistencies, which readily enumerates local sources of inconsistencies even when the requirement model is inconsistent.

We follow (and extend) the OpenReq datamodel[5] [15]. Hence, any issue is considered as a *Requirement* that is characterized, among others, by *priority* (integer, smaller number is higher) and *effort* (integer, e.g., in hours), *status* such as 'planned' or 'complete' as well as requirement *text*. A requirement can be assigned to a *Release*. A Release is characterized by startDate, releaseDate, *capacity* (e.g., in hours) and *version* string. The version strings conform to the common notation: E.g., '4', '4.1' and '4.1.12' are version strings. They can be amended with prefixes and suffixes, e.g. ABC-4.12.1RC1 represents Release Candidate 1 of the version 4.12.1 of product ABC[6].

Dependencies are binary relationships between two requirements. The types of dependencies with clear semantics are summarized in Table 1. In the table, rel_{ra} and $prio_{ra}$ specify the assigned release and priority of requirement ra, respectively. Assignment to release 0 means that the requirement is not assigned to any release. Many of the dependencies are similar to those identified in [9]. The dependency duplicates(ra, rb) is managed in pre-processing by collecting all dependencies of rb to ra.

The compositional structure of requirements is expressed as *decomposition* dependencies. This part-of hierarchy of requirements seems to be typically 4 levels of depth at maximum. For instance, Epics can have user stories, and user stories can have task.

2.3 Solution Functionality

The current main functionalities for requirement engineering are consistency checks and diagnosis services as well as computation of transitive closure. The user interface for and visualization of dependencies by *OpenReq Issue Link Map*[7] [12] is vital for practical usage but not the focus here.

Transitive closure service computes a transitive closure of a Requirement in focus of analysis by following all links in breath-first manner up to the specified *depth* in terms of the number of dependencies followed. By adjusting the desired depth, different contexts of analysis can be formed. For releases, the current implementation calls the service for each requirement of the release and combines the results.

Consistency check analyzes a defined contexts of analysis formed by a set of requirements and their dependencies, priorities, and releases. The following aspects are checked: Each binary dependency must satisfy the semantics of the dependency as defined in Table 1. Here, the assigned release and the priority of each requirement is taken into account. If effort consumption is specified, the sum of efforts of requirements assigned to a release must be less or equal than the capacity of the release. The analysis reports aspects such as inconsistent relationships and resource consumption per release. Both human-friendly messages and machine-friendly JSON data fields are included in the response.

Diagnosis can be optionally performed in conjunction of a consistency check. Diagnosis attempts to provide a 'repair' by removing requirements or dependencies. Requirement removal is justified especially when capacity consumption is excessive. It is also possible that assignments or dependencies have been performed in a faulty manner. Therefore, a diagnosis (1) Can consider requirements as faulty; (2) Can consider relationships as faulty; and (3) Can consider both requirements and relationships as faulty.

If all the elements proposed by a diagnosis (relationships, dependencies) are removed (requirement is unassigned, represented by assigning it to release 0), a consistent release plan is achieved. Diagnosis can also fail. For example, removing only relationships cannot fix excessive resource consumption. Diagnosis is based on the FASTDIAG algorithm [8].

[4] release-based analysis is in early stages of development
[5] https://github.com/OpenReqEU/openreq-ontology
[6] We apply the Maven Comparable Versions: https://maven.apache.org/ref/3.6.0/maven-artifact/apidocs/org/apache/maven/artifact/versioning/ComparableVersion.html
[7] https://api.openreq.eu/openreq-issue-link-map

Table 1. Semantics of Dependencies

Dependency	Closest type in [9]	Semantics	Description
excludes(ra, rb)	atmostone(rel_{ra}, rel_{rb})	$rel_{ra} = 0 \vee rel_{rb} = 0$	at most one out of $\{ra, rb\}$ has to be assigned to a release
incompatible(ra, rb)	different(rel_{ra}, rel_{rb})	$rel_{ra} \neq rel_{rb} \vee rel_{ra} = 0 \vee rel_{rb} = 0$	$\{ra, rb\}$ have to be implemented in different releases
requires(ra, rb)	weakprecedence(rel_{rb}, rel_{ra})	$rel_{ra} = 0 \vee (rel_{rb} \leq rel_{ra} \wedge rel_{rb} > 0)$	rb must be implemented before ra or in the same release, or ra is not in any release
implies(ra, rb)	strongprecedence(rel_{rb}, rel_{ra})	$rel_{ra} = 0 \vee (rel_{rb} < rel_{ra} \wedge rel_{rb} > 0)$	rb must be implemented before ra or ra is not in any release
decomposition(ra, rb)	(none)	$rel_{ra} = 0 \vee (rel_{ra} > 0 \wedge rel_{rb} > 0 \wedge (rel_{rb} \leq rel_{ra} \vee prio_{rb} > prio_{ra})$	Whole ra is not complete without part rb: rb must be implemented at the same release or before ra or rb has a lower priority so it can be assigned to a later release. A better name would be haspart(ra, rb)

For example, assume that Release 1 of capacity 3 (hours) has assigned requirements REQ1 (effort:2h) and REQ2 (2h). Release 2 of capacity 4 has REQ3 (3h) and there is a dependency excludes(REQ1, REQ2). Analysis and diagnosis results would include, among others, (white space modified):

```
{"response": [{
"AnalysisVersion": "analysis",
"AnalysisVersion_msg": "Analysis and consistency check",
"Consistent": false,
"Consistent_msg": "Release plan contains errors",
"RelationshipsInconsistent": [{
    "From": "REQ1", "To": "REQ2", "Type": "excludes"}],
"RelationshipsInconsistent_msg":
"Relationships that are not respected (inconsistent): rel_REQ1_excludes_REQ2",
"Releases": [{ ** Release 0 omitted||() ...
    "Release": 1,     "Release_msg": "Release $1$",
    "RequirementsAssigned": [
      "REQ2",
      "REQ1"],
    "RequirementsAssigned_msg": "Requirements of release: REQ2, REQ1",
    "AvailableCapacity": 3,     "CapacityUsed": 4,     "CapacityBalance": -1,
    "CapacityUsageCombined_msg": "Capacity: available 3h, used 4h, remaining -1h"},
...
{"AnalysisVersion": "reqdiag",
"AnalysisVersion_msg": "Requirements diagnosis",
"Consistent": true,
"Consistent_msg": "Release plan is correct",
"Diagnosis": {
  "DiagnosisRequirements": [
    "REQ1"  ],
  "DiagnosisRelationships": []},
"Diagnosis_msg":
"Diagnosis: remove these requirements (REQ1) AND these relationships ((none) )",
...
{"Release": 1,
    "Release_msg": "Release $1$",
    "RequirementsAssigned": ["REQ2"],
    "RequirementsAssigned_msg": "Requirements of release: REQ2",
    "AvailableCapacity": 3, "CapacityUsed": 2, "CapacityBalance": 1,
    "CapacityUsageCombined_msg": "Capacity: available 3h, used 2h, remaining 1h"},
...
```

The Diagnosis of Requirements would suggest removing REQ1. Updated capacity calculations and resulting release assignments are reported. Diagnosis of only relationships cannot succeed, because of the excess capacity.

2.4 Solution Architecture and Implementation

We have implemented the approach as a service-based system consisting of independent services[8], which in practice operate in a choreographic manner combining the pipe-and-filter and layered architectural styles (Fig. 2). The services collaborate through message-based interfaces following REST principles.

The basic services realize the concepts described above by two services: *KeljuCaaS* and *Mulperi*. KeljuCaaS is a Configurator-as-a-Service, whose responsibility is to provide analyses for models that it receives from Mulperi. Currently, KeljuCaas provides functionality described in Section 2.3 based on information described in Section 2.2. For consistency check, KeljuCaas has a procedural component that checks the model for inconsistencies and reports them. These inconsistencies may result from dependencies between requirements including their assignments to releases, priority violations, or requirement efforts exceeding the capacity of the release. For diagnosis, KeljuCaas converts the release plan into a Constraint Satisfaction

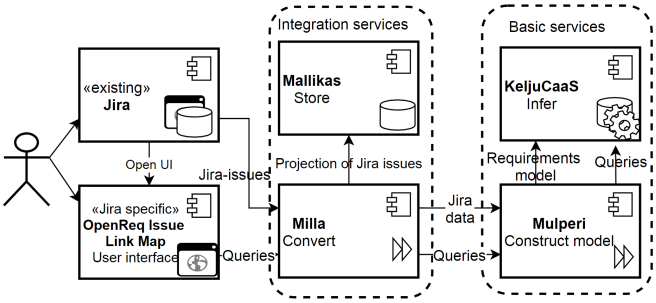

Figure 1. The architecture of the system.

Problem and uses the Choco Solver [14] and FastDiag [8]. The additional functionality of KeljuCaas is to form and maintain a graph containing all received requirements for caching purpose for large data sets. The graph can then be searched for related requirements in a transitive closure of a single requirement for the specified depth that is used for visualization and analysis. *Mulperi* service operates as a pipe-and-filter facade component to transform data and format data to KeljuCaaS and provides its answers back to the caller. For example, in a case of a small data, Mulperi can directly send data to KeljuCaaS, whereas in a case of large data such as in Qt's Jira, Mulperi differentiates functionality to send data to KeljuCaaS to construct the graph and any requested consistency check first queries this graph for a transitive closure for desired depth that is then sent for consistency check. The reason for separating the functionality of Mulperi from KeljuCaaS is to keep inference in a more generic service.

The integration services provide integration with existing requirements management systems, specifically with Qt's Jira. The key facade and orchestrator service is called *Milla*. Milla imports Qt's Jira issues as JSON from the Jira's REST interface and converts them into Java objects. These objects are sent from Milla to *Mallikas* database for caching storage as well as to Mulperi for processing. Milla is also able to fetch new or modified issues from Jira to keep data up to date. Mallikas is a simple database for storing Qt's Jira issues as objects. It uses the H2 database engine and Java Persistence API to cache the data. This improves performance and avoids constant access to Jira.

The user interface is provided with OpenReq Issue Link Map[9] (Fig. 2). The user interface shows a 2D diagram of dependencies from the desired issue by selected depths. An issue can be searched or clicked on the diagram. On the right, tabs separate basic information,

[8] EPL licensed https://github.com/OpenReqEU

[9] The demo version is available through https://openreq.eu/tools-data/

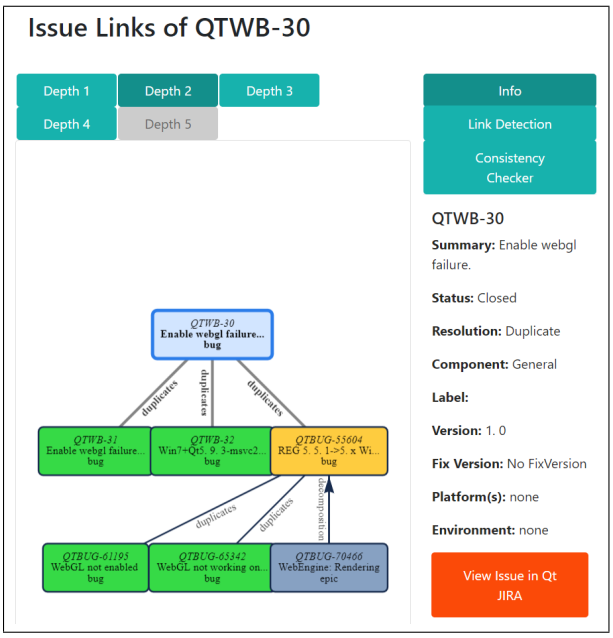

Figure 2. A screen capture of OpenReq issue link map.

dependency detection, and the results of consistency check.

3 Evaluation Context: The Qt Company & Jira

We demonstrate practical application of our approach in realistic settings and evaluate performance using Jira of the Qt Company. The Qt Company is a public company having around 300 employees and the headquarters in Finland. Its product, Qt[10] is a software development kit that contains a software framework and its supporting tools. The software framework is targeted especially for cross-platform mobile applications, graphical user interfaces, and embedded application development. A well-known application example using Qt is the Linux KDE desktop environment but most of today's touch screen and embedded systems with a screen use Qt.

3.1 Jira's Data Model at the Qt Company

All requirements and bugs of Qt are managed in the Qt's Jira[11] issue tracker that has been in use for over 15 years. Jira[12] is a widely used issue tracker that provides many issue types and a lot of functionality, especially for individual issue management. All product planning at Qt is performed using Jira, despite attempts to integrate with roadmapping tools. Qt has configured Jira for its needs. In the following, we describe the Jira data as applied at Qt.

Jira is organized into projects consisting of issues. The issues are divided into different *issue types* as shown at the top row of Table 2. A *bug* refers basically to any deficiency found from the existing software. However, the difference between a deficiency and a new feature is not always clear. A bug report can request also new features. *Epic*, *user story*, *task* and *suggestion* each refer to new development ideas or features. *Change requests* are used infrequently without clear purpose in most projects. A task actually differentiates between a task, sub-task, and technical task but there are no clear guidelines of use and the usage is not consistent. Thus, we do not differentiate between different task types.

[10] https://www.qt.io/
[11] https://bugreports.qt.io
[12] https://www.atlassian.com/software/jira

Table 2. The number of different issues types in Qt's Jira.

Project	total	bug	epic	user story	task	sugg-estion	change request
QTPLAYGROUND	15	11	0	0	0	4	0
QTWB	23	16	0	1	3	3	0
QTSOLBUG	193	122	0	0	8	63	0
QTSYSADM	261	16	0	0	242	2	0
QTJIRA	280	162	0	2	39	77	0
QSR	399	123	6	34	229	7	0
QDS	558	265	12	26	195	60	0
QTVSADDINBUG	629	514	0	21	14	80	0
QTWEBSITE	676	519	5	0	21	121	0
AUTOSUITE	871	330	67	159	298	17	0
PYSIDE	890	754	0	39	41	56	0
QTCOMPONENTS	1144	617	9	186	293	39	0
QTIFW	1266	931	2	12	119	202	0
QBS	1397	955	6	4	226	206	0
QTMOBILITY	1926	1538	0	0	93	149	146
QTQAINFRA	2635	915	29	120	1444	127	0
QT3DS	3292	1685	52	165	1227	163	0
QTCREATORBUG	21217	16975	3	76	1163	2979	21
QTBUG	74287	58583	223	623	6182	8636	40
Total	111959	85031	414	1468	11837	12991	207

The issue types define common *properties as name-value pairs*, customizable by issue type. The property values can be text, such as for a title and description; an enumerated value from a closed set, such as for priority; an enumerated value from an editable and extending set, such as for release numbers or users; or a date. Each issue can have comments. The change history of the issue is logged. The relevant properties in this context are *priority* and *fix version*. Priority has predefined values from P0 to P6. P0 'blocker' is the highest priority and P6 is the lowest priority. A fix version refers to the release in which the issue has been or will be completed and adheres to maven convention described above.

Jira has six different directed dependency types knows as *links*: *duplicate*, *require*, *replace*, *results*, *tests*, *relates* (cf. the top row of Table 3). Only 'requires' and 'duplicate' have a clear semantics. The other dependency types are used non-uniformly.

In addition, Jira has *decomposition* (parent-child) relationship. `Issues in Epic` is used to add any other type of issues than epic as child to an epic. `Sub-task` relations are used to add tasks as child to other issues than tasks. However, the semantics is the same for all decomposition relationships even though the name differs. As the result, issues can have an up to three level compositional hierarchy.

The resulting rules regarding the dependencies are the following: All child issues, which have the same or higher priority, must not be assigned to a later release; any required issue must not have a later release or lower priority; and all links from a duplicated issue are inherited by the duplicate issue

3.2 Data Quantity and Characteristics

The data in Qt's Jira is divided into public and private parts. The private part includes a couple of thousand issues of confidential customer projects and Qt's strategic product management issues. We focus here only to the public part because it is significant enough as it contains most (roughly 98%) of the issues, and describes most technical details.

Qt Jira is divided into 19 projects (Table 2). 'QTBUG' is the main

Table 3. The number of different dependency types in total and internal pointing to an issue in the same project.

	Total								Internal							
Project	total	part	duplicate	require	replace	results	tests	relates	total	part	duplicate	require	replace	results	tests	relates
QTPLAYGROUND	0	0	0	0	0	0	0	0	0 (0%)	0	0	0	0	0	0	0
QTWB	9	1	6	0	0	0	0	1	2 (22%)	0	2	0	0	0	0	0
QTSOLBUG	13	0	0	2	6	0	0	4	7 (53%)	0	0	1	6	0	0	0
QTSYSADM	11	0	0	0	2	6	0	2	9 (81%)	0	0	0	1	6	0	2
QTJIRA	17	0	1	2	8	0	0	6	12 (70%)	0	0	0	7	0	0	5
QSR	364	306	1	48	0	2	0	7	333 (91%)	284	1	41	0	1	0	6
QDS	265	191	2	45	0	5	0	22	205 (77%)	172	1	19	0	1	0	12
QTVSADDINBUG	77	2	26	21	7	4	2	15	73 (94%)	1	26	21	7	2	2	14
QTWEBSITE	21	8	0	2	0	0	0	8	16 (76%)	8	0	1	0	0	0	7
AUTOSUITE	326	255	4	36	2	9	0	20	259 (79%)	203	3	27	1	6	0	19
PYSIDE	147	14	28	26	2	12	0	65	127 (86%)	13	25	23	1	10	0	55
QTCOMPONENTS	311	169	0	66	10	35	0	31	265 (85%)	169	0	29	10	32	0	25
QTIFW	248	51	34	33	60	9	0	53	140 (56%)	41	30	6	29	4	0	30
QBS	290	57	17	67	36	13	0	96	237 (81%)	50	12	50	28	10	0	87
QTMOBILITY	299	169	0	29	26	33	0	42	268 (89%)	169	0	17	19	26	0	37
QTQAINFRA	1221	566	37	384	23	51	1	152	712 (58%)	421	23	152	19	19	0	78
QT3DS	1816	1170	11	231	6	173	0	225	1705 (93%)	1170	7	189	6	159	0	174
QTCREATORBUG	4056	592	530	343	1198	221	8	1141	2975 (73%)	366	478	172	956	131	4	868
QTBUG	15366	3567	1858	3371	1280	1063	11	4152	13767 (89%)	3390	1826	2880	1009	913	6	3743
Total	24857	7118	2555	4706	2666	1636	22	6042	21112 (84%)	6457	2434	3628	2099	1320	12	5162

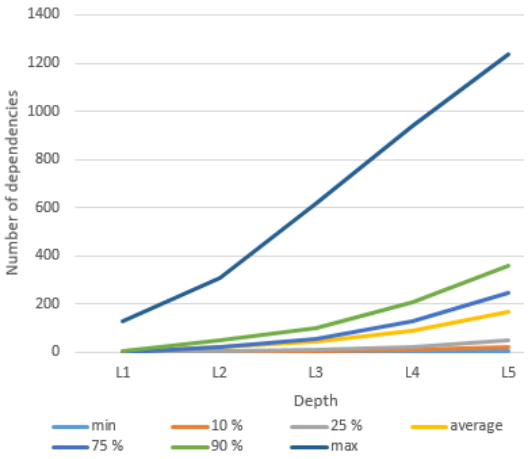

Figure 3. The number of dependent issues at different depths.

project covering the Qt framework itself and 'QTCREATORBUG' is the IDE for the framework. These two projects are the largest but also the most relevant ones. The other projects are much smaller and some of them are even inactive. The number of different issue types and dependencies are shown in Tables 2 and 3, respectively. It is also noteworthy that while most dependencies are internal to a project it is not uncommon to have dependencies between projects.

The dependencies form a set of graphs between the issues through their relationships transitively. Figure 3 illustrates the sizes of such graphs in small depths. In the data, there is one graph that contains 6755 issues as its nodes and the greatest depth in this graph is 52 dependencies (edges). The remaining graphs are significantly smaller, the next largest ones containing 376, 164, 118, 114, and 91 issues, and depths of 29, 21, 5 and 8 dependencies, respectively. As Figure 3 illustrates, the number of issues can grow relatively quickly when depth grows. There are also small graphs: 9431 and 5488 issues participate in a graph with only one and two other issues that can include private issues. 84497 (75%) of issues are *orphans* meaning that they do not have any explicit dependency to another issue.

QTBUG has the most rigorous release cycle that we describe as follows. In total, there are 164 releases, out of which 26 are empty releases without any issues. We did not investigate the reasons for empty releases but it is possible that issues have been moved to some other release and release is not done. The average and median number of issues in a non-empty release is 194 and 122, respectively. Three releases have a large number of issues: 5.0.0/2142, 4.8.0/882, and 4.7.0/1571. Since 5.0.0. released December 2012, Qt5 has already 111 releases. Qt 6.0.0 is planned for the November 2020.

4 Performance Evaluation

4.1 Approach for performance testing

We tested end-to-end performance of our system with consistency checks and diagnosis, because they concern main functionality and are potentially computationally heavy.

Performance tests for an individual issue used each issue of a project in turn as a starting point *root issue*. The transitive closure of different depths (1, 2, ...) was calculated for the root issue forming a *test set*. The test were carried out to all issues at all existing depths. As the depth increased, the number of existing graphs at that depth decreased resulting in carrying out test to different sub-graphs of a small number of large graphs. Issues without dependencies were filtered out. Consistency check and, depending on the test, diagnosis were performed for the test set with a timeout. To limit execution time required by testing, testing of the root item with even greater depths was not performed after the first time-out was encountered.

The test were ran as Unix-like shell scripts for the system running in the localhost. The system exhibits overhead caused by the service architecture. In order to estimate the overhead of architecture and testing for the response times, we carried out consistency check for a set of 1000 issues that do not have any dependencies. The time required for the consistency check should be minimal. The response

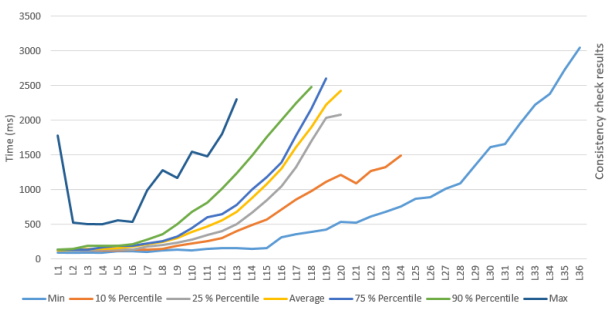

Figure 4. Time for consistency check using 3s timeout. Max at depth 1 is probably an error.

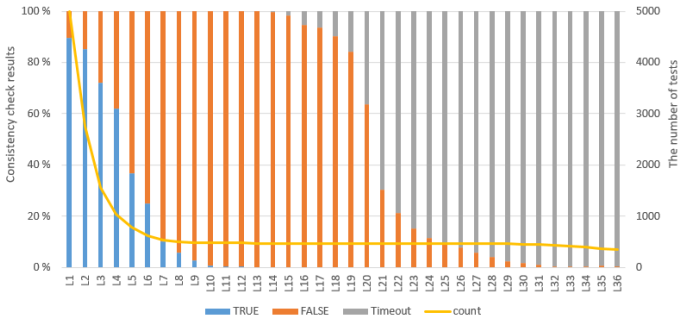

Figure 5. Consistency check results in percentages by depth using 3s time-out. The yellow line shows the count of the executed test at each depth.

times were: average 128ms, minimum 103ms, maximum 216, and standard deviation 9ms.

The tests were carried out using a 64bit Windows 10 laptop having Intel Core i5-7200 CPU @2.5GHz and 16GB RAM. The tests were executed typically over the night but the computer was also occasionally used at the same time for office tasks such as text editing especially when execution had not yet completed. The tests used the data retrieved from Qt Jira in May 2019.

4.2 Consistency Check of an Individual Issue

The first performance test measures consistency checks without diagnosis as described in Section 4.1 for QTCREATORBUG using 3ms timeout. The largest graph in QTCREATORBUG has the maximum depth of 48[13] and this graph contains 6755 issues out of which 466 are in QTCREATORBUG. We carried out 18950 consistency checks tests successfully while 7789 tests caused timeout—or would have been scheduled for the same issue at a greater depth than the first timeout. After level 36, which contained 300 tests, all tests caused timeout and tests at the greater depth are omitted from below.

Figure 4 exhibits the time required for the consistency check and Figure 5 the respective results of the consistency check. The lines take into account timeouts: For example, 75% percentile line ends at the depth 19 when over 25% tests results cause timeout because the percentile cannot be calculated anymore. The first timeouts took place at depth 14 for two items that had 4350 and 4253 issues in their graphs. In fact, the smallest graph that contained a timeout was 3816 issues. Until depth of 5, over 60% of test sets are consistent but adding depth quickly decreases the share of consistent test sets.

As a comparison, the same test script was ran for all Jira data using another laptop running Cubbli Linux (Ubuntu variant of University of Helsinki) having Intel Core i5-8250U CPU @1.60GHz and 16 GB RAM. These test were carried out during a weekend when the computer was otherwise idle. The tests took about 25 hours. These tests used another snapshot of all data in Jira, which was about half year old data downloaded for development and testing purposes. For example, the largest graph of maximum depth 47 in this data consisted of only 3146 issues. That is, some of the graphs were apparently combined later by new dependencies. 171498 tests were executed. No timeout occurred and the longest execution time was 2652ms.

[13] The form of the graph is such that using any QTCREATORBUG node as starting points does not create the maximal depth of 52, i.e., the nodes are not at the 'outer front' of the graph.

4.3 Diagnosis and Consistency Check of an Individual Issue

The second test performed consistency checks with diagnosis as described in Section 4.1 for QTCREATORBUG. All three diagnoses are invoked in the case of an inconsistent test set. As the diagnosis is carried out only for inconsistent issue graphs, we excluded consistent graphs. The results of the execution time with respect to the number of dependencies are much worse than without diagnoses. The 3000ms time-out is quite tight, because the system performs three separate diagnoses, leaving, on the average, slightly less than one second for each. The results (Fig. 6) show that the timeouts start already from depth 3. While at depth 7 only 17% resulted in timeout, the following depths timeout became frequent (depth 8/65%, 9/81%, and 10/89%), and at level 18 all resulted in a timeout. Inspecting the graph sizes, two smallest graphs causing a timeout were only 26 and 75 issues. This may have been caused by computer overloading for other use. Starting from the third smallest graph resulting a time-out at the size of 129 issues, timeouts become frequent and the 30th smallest graph causing a timeout has only 149 issues.

4.4 Consistency Check of a Release

Performance tests for a release follow the above scenario of individual tests except that a release consists of a set of issues rather than a single issue. Therefore, a root can consist of several issues. In the current implementation, a graph of each single issue of a release was fetched. All graphs were sent to consistency checker. For any larger release consisting of several issues, significant overhead was caused

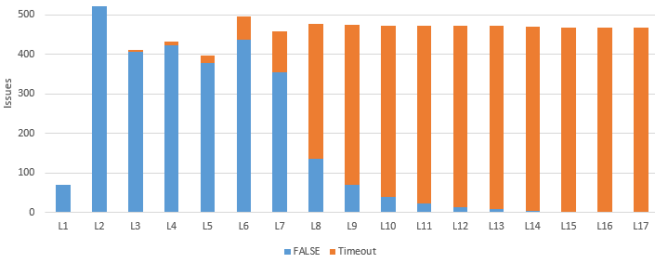

Figure 6. Consistency check and diagnosis of project QTCREATORBUG: Number of inconsistent and time-out results per depth.

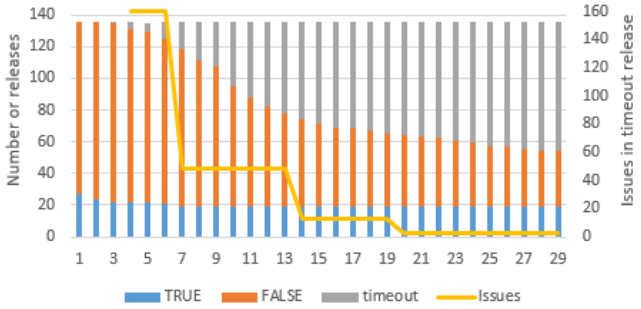

Figure 7. Consistency check results for QTBUG's releases using 10s timeout and the smallest number of issues in the timeouting release.

by generating all the graphs and repeating same data over and over. Here, we applied 10s timeouts. Consistency checks were performed for all non-empty releases of QTBUG. The five largest releases containing over 700 issues caused an error at the REST interface of the service due to the number of parameters given as root. Fig. 7 illustrates the results. The first timeout occurred at depth 3 with a release containing 610 requirements. It is omitted from Fig. 7 for readability.

5 Analysis of performance results

The performance tests were carried out using the Jira data of the Qt Company that we consider as a large empirically valid data set. Although more complex synthetic data could be constructed, the Jira data forms a more realistic and solid base for performance testing for fast enough response times.

The dependencies form graphs of issues as transitive closures that vary in their depths and sizes. We used the depth from a selected issue as a variable to vary the sizes of graphs. A user cannot initially know the size of the graph at the context of analysis. Thus, means to limit the size are required, and limiting the depth is a natural approach. The majority of issues remain orphans. It is noteworthy that there is only one very large graph of over 6000 issues, a few graphs of hundreds of issues, and several graphs of tens of issues. The releases merge these graphs whenever they include issues from different graphs.

For all issues of Qt, the performance is adequate for performing consistency analysis of the neighborhood of an issue interactively, practically even with any depth. The timeouts started to appear for graphs of around 4000 issues and the depth of 14. Diagnosis is more computationally heavy but it performs quite well until depth 7 which is adequate taking into account that small (3 s) timeout was applied, three diagnoses were performed and the number of issue is in average close to 200 already at depth 5.

Consistency checks for releases work sufficiently well until depth 6 in most of the cases. When the context of analysis is a release, the performance results are probably significantly too pessimistic: This new feature has no direct support for calculating the transitive closure. Instead, individual transitive closures of the issues of the release are calculated and finally combined. This leaves significant potential for optimizing.

We applied mainly a three second time-out in performance testing to shorten test duration with large data sets. In our view, analysis and diagnosis of a whole release justifies, also from user point of view, a much longer time-out value in the worst case scenarios.

6 Discussion

Validity of this work is exposed to some threats. First, the tests were performed with all project data available. Therefore transitive closures span several projects. This has a side effect on the reliability of the results: Because versions are not comparable across projects, cross-project dependencies may be consistent or inconsistent in a faulty manner. Therefore, we decided against reporting the number of erroneous dependencies. Because versions are not comparable across projects, many dependencies would be considered as not satisfied although they are satisfied, and vice versa.

Second, it is noteworthy to observe that the results for the number of dependencies greater than 100 are done for the different sub-graphs of the few large graphs. Some of these sub-graphs are very similar as the test are done for all possible sub-graphs. In particular, our test scripts analyzed numerous sub-graphs of the largest graph with 6755 dependencies as a different issue of the graph was selected as the root issue. A preliminary inspection did not indicate that this large graph or its sub-graphs would otherwise differ from other graphs but this would deserve a more thorough analysis.

Third, we did not control the test environment rigorously. Especially other software running at the same time probably affected the results. Even the computer was a normal office laptop rather than a proper server computer.

Despite the above mentioned non-trivial threats to construct validity, our view is that the big picture of results is still valid, although some details might be incorrect. In other words, our view is that the approach performs well enough for practical use at Qt. We believe that this can be generalized to other contexts too.

Future work is required to more realistically gain benefits from the approach and the system developed.

The visualization tool should be extended so that it can highlight inconsistent dependencies and also show diagnosis results graphically. In our view, these extensions will make showing diagnosis results to stakeholders much more intuitive than current textual descriptions. Empirical studies on the benefits of the approach are best performed with this support at hand.

In addition to individual issues and releases of different depths, other context of analysis can be relevant. For example, small projects can potentially be relevant contexts of analysis, such as QT3DS that has around 3300 requirements and is under active development. Similarly, a specific component or domain, such as Bluetooth or all networking, could form a context of analysis or a be used as a filtering factor similarly as depth.

The visualization tool currently can visualize the neighbourhood of a requirement up to 5 levels of depth. When all 5 levels exist, the graph has on the average 170 dependencies. This would suggest that 5 levels is enough. However, the minimum is 5 issues and the 10% percentile has only 21 issues. The ability to constrain the scope of the graph is important because too large graphs may not be useful for stakeholders and smaller contexts of analysis are easy for consistency checks and perform well also with diagnosis. Instead of a fixed depth limit, it might be practical to be give as parameters any desired depth and an upper bound on the number of issues to retrieve for visualization and analysis.

As the issue tracker data is manually constructed by different stakeholders, not all dependencies are marked. We are studying the detection of dependencies via natural language processing. The challenges with Qt's Jira data in many approaches is that they can propose too many dependencies, they are computationally heavy and the semantics of only duplication dependency is easy to detect. In consis-

tency check, proposed dependencies should probably not be treated as equal to existing ones unless manually accepted by a user.

We currently consider the whole database of issues but do not take into account many of the issue properties. For example, status, resolution, creation date, and modification date could be taken into account as filters. For example, inconsistent dependencies among completed, very old issues may be irrelevant, even if they are broken.

Besides Jira, a tight integration to other trackers could be added by developing similar integration services. However, it is already possible to communicate through a JSON-based REST interface.

We focused primarily on the technical approach and its performance. The user point of view was considered only in terms of the relevant size of issue graphs. While users should be studied in more depth, also the technical proposals deserves user studies. Currently the system calculates and provides all three different diagnoses. If the user is interested in only one of them, a significant increase of performance would be achieved simply by performing only the desired diagnosis. Besides repairing a release plan by removing inconsistent requirements or relationships, future work could consider reassigning requirements to other releases. However, such decisions are at the heart of product management decisions. There are often aspects in decision making of product management that are not easy to formalize. It may often be more important to get understanding about the problem than get less-than-solid proposals of repair.

7 Conclusions

This work is a contribution in the area of KBC: we assist in producing consistent, connected configurations of requirements, apply techniques of KBC to a relatively new domain, and apply our approach to a large set of real industrial data providing evidence that the approach is viable. We identified major requirements and developed an approach that can support product management, requirements engineering and developers in practically important use cases in contexts where large, inconsistent bodies of requirements are typical. Empirical evaluation shows that the approach scales to usage in large projects, but future work for improving performance in some use cases is still required.

The approach builds on considering a body of requirements as a configuration of requirements that should be consistent, but it often is not. Via neighbourhoods of different depth from a requirement or a release, we support different sizes of contexts of analysis. Contexts of a reasonable size facilitate solving identified problems.

With the support developed, developers can conveniently visualize related requirements and their dependencies; a requirement engineer can identify problematic dependencies and attempt to remedy them; and a product manager can more easily manage the consistency of a release. The performance of the tool is adequate for these tasks, except that the diagnosis of a whole release needs further work and modifications to the REST interface that cannot currently accommodate releases of 700 issues of more.

Value in real use is highly plausible but demonstration requires tighter integration with a developed visualization tool, which would enable experiments with real users. Our work can be seen as (continuation of) extending Knowledge Based Configuration to requirements engineering and product management.

ACKNOWLEDGEMENTS

This work is a part of OpenReq project that is funded by the European Union's Horizon 2020 Research and Innovation programme under grant agreement No 732463. We thank Elina Kettunen, Miia Rämö and Tomi Laurinen for their contributions to implementation.

REFERENCES

[1] Philip Achimugu, Ali Selamat, Roliana Ibrahim, and Mohd Naz'ri Mahrin, 'A systematic literature review of software requirements prioritization research', *Information and Software Technology*, **56**(6), 568–585, (2014).

[2] David Ameller, Carles Farré, Xavier Franch, and Guillem Rufian, 'A survey on software release planning models', in *17th International Conference Product-Focused Software Process Improvement (PROFES)*, pp. 48–65, (2016).

[3] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. Natt och Dag, 'An industrial survey of requirements interdependencies in software product release planning', in *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pp. 84–91, (2001).

[4] John Wilmar Castro Llanos and Silvia Teresita Acuña Castillo, 'Differences between traditional and open source development activities', in *Product-Focused Software Process Improvement*, pp. 131–144, (2012).

[5] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose, 'Predicting the delay of issues with due dates in software projects', *Empirical Software Engineering*, **22**(3), 1223–1263, (Jun 2017).

[6] Åsa G. Dahlstedt and Anne Persson, *Engineering and Managing Software Requirements*, chapter Requirements Interdependencies: State of the Art and Future Challenges, 95–116, Springer, 2005.

[7] Maya Daneva and Andrea Herrmann, 'Requirements prioritization based on benefit and cost prediction: A method classification framework', in *34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 240–247, (2008).

[8] A. Felfernig, M. Schubert, and C. Zehentner, 'An efficient diagnosis algorithm for inconsistent constraint sets', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **26**(01), 53–62, (2011).

[9] Alexander Felfernig, Johannes Spöcklberger, Ralph Samer, Martin Stettinger, Müslüm Atas, Juha Tiihonen, and Mikko Raatikainen, 'Configuring release plans', in *Proceedings of the 20th Configuration Workshop, Graz, Austria, September 27-28, 2018.*, pp. 9–14, (2018).

[10] S. Gregor, 'The nature of theory in information systems', *MIS Quarterly*, **30**(3), 611–642, (2006).

[11] Laura Lehtola, Marjo Kauppinen, and Sari Kujala, 'Requirements prioritization challenges in practice', in *5th International Conference Product Focused Software Process Improvement: (PROFES)*, pp. 497–508, (2004).

[12] Clara Marie Lüders, Mikko Raatikainen, Joaquim Motger, and Walid Maalej, 'Openreq issue link map: A tool to visualize issue links in jira', in *IEEE Requirements Engineering Conference*, (2019 (submitted)).

[13] Klaus Pohl, *Process-centered requirements engineering*, John Wiley & Sons, Inc., 1996.

[14] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca, *Choco Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. www.choco-solver.org, 2016.

[15] Carme Quer, Xavier Franch, Cristina Palomares, Andreas Falkner, Alexander Felfernig, Davide Fucci, Walid Maalej, Jennifer Nerlich, Mikko Raatikainen, Gottfried Schenner, Martin Stettinger, and Juha Tiihonen, 'Reconciling practice and rigour in ontology-based heterogeneous information systems construction', in *The Practice of Enterprise Modeling*, pp. 205–220, (2018).

[16] Mikael Svahnberg, Tony Gorschek, Robert Feldt, Richard Torkar, Saad Bin Saleem, and Muhammad Usman Shafique, 'A systematic review on strategic release planning models', *Information and Software Technology*, **52**(3), 237 – 248, (2010).

[17] R. Thakurta, 'Understanding requirement prioritization artifacts: a systematic mapping study', *Requirements Engineering*, **22**(4), 491–526, (2017).

[18] A. Vogelsang and S. Fuhrmann, 'Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study', in *21st IEEE International Requirements Engineering Conference (RE)*, pp. 267–272, (2013).

[19] Roel J Wieringa, *Design Science Methodology for Information Systems and Software Engineering*, Springer, 2014.

[20] H. Zhang, J. Li, L. Zhu, R. Jeffery, Y. Liu, Q. Wang, and M. Li, 'Investigating dependencies in software requirements for change propagation analysis', *Information and Software Technology*, **56**(1), 40–53, (2014).